superuser community blog

Linux permissions demystified

April 22, 2011 by flibs. 8 comments

I have seen a large number of questions on Super User recently all around the same topic of Linux and Unix file permissions. For example:

unix file permissions

> File permissions in UNIX are frequently specified as an octal number. Why is octal the preferred base for this purpose?

… and …

Linux file permissions

> I own a particular file on a Linux system. I would like to give 2 groups (accounting, shipping) read access and only read access, and 3 users(Mike, Raj and Wally) write access and only write access. How can I accomplish this?

In a world of Windows where file permissions can be granted on a per-user basis, Linux and Unix permissions seem to be very hokey and restricted.

Well, let me tell you, they're not. For such a seemingly basic arrangement they are an incredibly powerful tool.

Most people forget that to do anything even remotely fancy with permissions in Linux you really have to couple them with groups. And you really can do some fancy things with them!

Basic Permissions

Let's start by looking at the basic permissions structure.

In Linux permissions are grouped into 3 main sections – *Owner*, *Group* and *World*. The *Owner* permissions define how the owner of the file can interact with the file, the *Group* permissions how people in the same group as the file can interact with it, and the *World* permissions how everyone else can interact with it.

These permissions are defined as a number between 0 and 7. They're in *Octal*. That's base 8 for the more programming literate amongst you.

An octal number is made up of 3 binary bits. In permissions the left-most bit is assigned to the Read permission, the middle bit to the Write permission, and the right-most to the Execute permission:

Read	Write	Execute	Octal
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Permissions in Linux are applied using the chmod command (short for Change Mode) like this:

```
$ chmod <octal> <filename>
```

So, for example, to allow the owner read and write, the group read, and stop anyone else from accessing the file at all, the octal would be (from the table above) 640:

```
$ chmod 640 myfile.txt
```

The ls -l command can inspect the file permissions that we have just set:

```
$ ls -l myfile.txt
-rw-r----- 1 matt matt 3675 2011-04-21 09:42 myfile.txt
```

This is a community blog for Super User. More community blogs are available at Blog Overflow.

Latest Articles

ISO files, optical drives and bootable flash drives
Geek on Sound (or.. does anyone really need a sound card these days?)
What to Do After Buying a New Laptop
Best of both worlds round 3: mSATA SSDs
Windows 8 on a VHD – Trying windows without the risk

Topics

Ask Different (12)
Backup & Restore (2)
Battle of the Giants (3)
Browsers (8)
Charity (1)
College (5)
Compression (3)
Computing (14)
Encryption (4)
Featured (14)
Filesystems (8)
Hard Drives (1)
Hardware (18)
History (1)
HTPC (2)
Interviews (7)
Memory (1)
Networking (10)
News (3)
Operating Systems (14)
Organizing (2)
Podcasts (12)
Productive Thursday (7)
Question of the Week (33)
Reviews (12)
Security (1)
Software (29)
Solid State Drives (9)
Super User (6)
Ubuntu (5)
Uncategorized (6)
Utilities (6)
Virtualization (3)

The string of characters on the left is the file permissions. Any permission that is not set is depicted as a –

No, your eyes aren't deceiving you – there really is one extra – there than you'd expect. The very first – is not a file permission, but a description of what the file is. A *d* in that position means the file is actually a directory. An *l* means it's a symbolic link, a *p* for a named pipe, etc. A – means that it's a normal, bog standard, file.

So splitting it down into groups of three we see: *-/rw-/r–/—* which is what we'd expect.

Immediately following the file permissions there is a 1 – this is the number of references to the file – hard links. We can ignore this for now. Next come two matt's. The first is the username of the file's owner, and the second is the group the file belongs to. Not the group the owner of the file belongs to as you might expect, the file itself belongs to a group.

Belong to a group?

The group ownership of the file can be changed using the *chgrp* command (short for Change Group):

```
$ chgrp users myfile.txt
$ ls -l myfile.txt
-rw-r----- 1 matt users 3675 2011-04-21 09:42 myfile.txt
```

However, you can only change a file's group ownership to that of a group you yourself are a member of unless you are root.

So now other people in the *users* group can read the file.

Adding people to a group is as simple as (as root):

```
# usermod -a -G <group> <username>
```

Be sure to include the -a flag or you will replace all the groups the user is a member of instead of appending the new group.

Groups definitions are usually stored in /etc/group, so you can look in there to see who is a member of which group. You can also use the *id* command to inspect a user:

```
$ id matt
uid=1000(matt) gid=1000(matt)
groups=1000(matt),4(adm),20(dialout),24(cdrom),46(plugdev),100(users),111(lpadmin),
    119(admin),122(sambashare),127(vboxusers)
```

You can see the full list of groups I belong to there.

So so far that's fairly simple and straight forward. You can easily allow groups of people to read specific files. Now let's get a little more fancy…

What if you want to provide a directory that members of a group can write to? You'd think that it would be a simple matter of making a directory and providing the group with write access. Well, let's take a look, shall we?

```
$ ls -ld shared
drwxrwx--- 2 fred users 4096 2011-04-21 10:00 shared
```

(note the -d flag to ls there – it says to list the directory itself, not the contents of the directory)

You can see the d at the beginning to show it's a directory, the read, write and execute to the owner (fred) and the group (users) and denied access to everyone else.

Directories treat the execute permission differently to files. For directories it actually means allow reading of the file list within the directory.

So let's create a file within that directory:

```
$ touch shared/testfile.txt
$ ls -l shared
total 0
-rw-r--r-- 1 matt matt 0 2011-04-21 10:03 testfile.txt
```

Yes, the user has managed to make a file in there. But you notice something? The group the file belongs to is the primary group of the user that made it. Now, the user could go back and use chgrp to change that, but that's somewhat cumbersome, yes?

Well, actually, right at the start of this post I lied to you. I said there were three groups of permissions. That's not strictly true. There's a fourth, extra special, group of permissions, which are the 'modifier' permissions. They don't actually define permissions themselves, but modify how the permissions of the other three sections work.

Windows (16)
Windows 8 Challenge (1)
WTFriday (10)

Getting sticky!

There are three special permissions: *SetUID*, *SetGID* and *Stciky*. They have different meanings when used on normal files compared to directories.

SetUID	SetGID	Sticky	Octal
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Normal files
The *SetUID* is used to change who a command is executed as. Some system-level commands are *SetUID* to root (The owner of the file is root and the *SetUID* flag is set) so that when the command is run it is elevated to have root level permissions. This is both exceedingly useful for system commands, and exceptionally dangerous – it's a prime candidate for hackers trying to obtain root permissions locally. Use with extreme care.

SetGID when applied to a file has no effect. Neither does the *Sticky* bit.

Dircetories
This is where the fun begins.

SetUID on a directory has no effect.

SetGID, however, is a whole different ball game. When a directory has the *SetGID* bit set and a file is created within that directory the group ownership of the file is automatically modified to be the group of the directory.

The *Sticky* bit means that only the owner of a file (or root) in this directory can delete or modify it. This is useful for shared temporary areas, like /tmp and /var/tmp.

So let's set the *SetGID* bit on the shared folder shall we?

```
(fred)$ chmod 2770 shared
ls -ld shared
drwxrws--- 2 fred users 4096 2011-04-21 10:11 shared
```

You see how the x permission for the group has been changed to an s now – the *SetGID* bit has been activated.

```
$ touch shared/anothertest.txt
$ ls -l shared
total 0
-rw-r--r-- 1 matt users 0 2011-04-21 10:20 anothertest.txt
-rw-r--r-- 1 matt matt  0 2011-04-21 10:11 testfile.txt
```

Et voila! The group for the new test file is set to users!

We're not quite there, however. The group members can't write to that file. A simple *chmod* of the file (660) would cure that, but again, that seems kind of clumsy.

There's one more command to introduce to you: *umask*

The *umask* defines the default permissions when a user creates a file. You might expect it to use the same format as *chmod*, but oh no, it's not quite that simple. It defines the permissions that get removed from the system default permissions when the file is created.

Most systems have a system-default set of permissions of 0666 for files (*rw-rw-rw-*) and 0777 for directories (*rwxrwxrwx*). The values in *umask* get subtracted from these numbers to give the new permissions:

```
$ umask
0022
```

You see that my umask is 0022 – so this gets subtracted from the default file permissions (not mathemtaically as such, but on a per-permission basis):

```
0 - 0 = 0
6 - 0 = 6
6 - 2 = 4
6 - 2 = 4
```

If any permission equates to a number less than 0 it is rounded up to zero, so a *umask* of 0027 would give the last number as 0 regardless of whether the system default had a 6 or a 7 in that place.

So my *umask* gives us a default permission of 0644, which equates to *rw-r–r–*, exactly what we are seeing above.

The *umask* command can be used to modify the user's *umask* at any time:

```
$ umask 0007
$ touch shared/finaltext.txt
$ ls -l shared
total 0
-rw-r--r-- 1 matt users 0 2011-04-21 10:20 anothertest.txt
-rw-rw---- 1 matt users 0 2011-04-21 10:28 finaltest.txt
-rw-r--r-- 1 matt matt  0 2011-04-21 10:11 testfile.txt
```

Perfect! The file is available for anyone in the users group to edit! The umask command can be put in your .bashrc file to keep the settings as you prefer them. Also, a number of file upload systems (like FTP servers) provide a setting for the umask when files get uploaded. This means that you can create an area into which any members of a group can upload files to – for example a shared website folder that anyone in the website managers group has access to.

Being individual

One final thing to mention:

There is a second way of setting the permissions in many Linux systems (and some Unix, but not all) which is a much easier way for changing individual permissions:

```
$ chmod <class><direction><permission> <file>
```

Where <class> is one of u (user),g (group) or o (other, or world), direction is a + or a -, and permission is one of r,w,x or s. This allows you to turn on and off individual permissions. For example:

```
$ chmod g+s shared
```

would turn on the SetGID bit of the shared directory.

```
$ chmod o-r myfile.txt
```

would remove the read permission from everyone else.

So you can see, the Unix file permissions system is actually considerably more powerful than it appears at first glance. It does take a bit of practice getting your head around it properly, but it is so surprisingly flexible.

Filed under Filesystems

Tagged: *nix, linux, Permissions, Unix

« Battle of the Giants: Microsoft vs. Google (Online Docs Edition) | Why you should forget about 4GiB of RAM on 32-bit systems and move on »

8 Comments

Subscribe to comments with RSS.

warren says:
April 22, 2011 at 1:17 pm

This is a great write-up of the basic *nix permissions model. But it does not address at all the second sample question (http://superuser.com/questions/161204/linux-file-permissions).

There are at least a couple questions on ServerFault that address this topic as well: http://serverfault.com/questions/84373/how-granular-is-too-granular-with-file-permissions & http://serverfault.com/questions/84370/what-filesystem-comes-closest-to-matching-ntfs-for-support-of-acls-and-highly-gr

(See also http://serverfault.com/questions/tagged/acl & http://serverfault.com/questions/tagged/permissions.)

flibs says:
April 23, 2011 at 10:24 pm

@warren – the only extra piece of knowledge you'd need for the second question is that if you want to specify multiple sets of attributes you will have to do it on separate hard-links to the file. Those can be created with the 'ln' command (ln myfile myfilelink). Then you can allow 1 group to have read access on "myfile" and 1 group to have write access on "myfilelink". Both files reference the same data, so changing one file will also change the other.

Unix file permissions does seem to be a commonly asked question all over the place – hence we decided to write an article about it.

Scott says:
October 20, 2014 at 8:45 pm

This is wrong. A hard link is a second directory entry that points to the same inode. Since all file metadata other than the name (including owner, group, mode (permissions number), and ACL) are in the inode, this means that a file and a hard link to that file ***cannot*** have different owners or groups.

warren says:
April 25, 2011 at 12:46 pm

@Matt – while an interesting solution, that also means every group needs to recall the name of their file to make edits .. seems like a lot more complicated than using ACLs.

flibs says:
April 26, 2011 at 9:16 am

@warren agreed, but ACLs aren't always available. This is meant to be a generic method that is available whether you are on Linux, BSD, SCO, AIX, Solaris …

Prashant says:
September 21, 2011 at 5:50 am

thank u sir, I got much useful information from you.

Mike B says:
November 17, 2011 at 11:36 pm

Nit:

-rw-rw—- 1 matt users 0 2011-04-21 10:28 finaltest.txt

Should be:

-rw-rw—- 1 matt users 0 2011-04-21 10:28 finaltext.txt

Good stuff

Swivel says:
December 2, 2016 at 7:19 am

Spelling error…

```
Getting sticky! There are three special permissions: SetUID, SetGID and Stciky. They have
different meanings when used on normal files compared to directories.
```

"Sticky" is spelt "Stciky" in the blog post (see snippet above)

Comments have been closed for this post